3.5 Simplifying If Statements

In the last section we introduced if statements, a powerful Python structure that allowed us to perform conditional execution of blocks of code. But as we'll see again and again in this course, expressive power comes with a cost: as our toolkit gets larger and the programming language features we use get more advanced, our programs also get larger and more complex; harder to read and reason about.

So every time we introduce a new part of the Python programming language, we'll also take some time to discuss not just what it can do, but also how to use it in structured ways that minimize the complexity we create by using it, and how to reason about its behaviour formally using tools from mathematical logic.

Computing booleans: when if statements aren't necessary

As our first example, consider the following function:

```python
def is_even(n: int) -> bool:
    """Return whether n is even (divisible by 2)."""
    if n % 2 == 0:
        return True
    else:
        return False
```

When we first learn about if statements, it is tempting to use them whenever we think of different "cases" of inputs, like even vs. odd numbers in this example. But remember that if statements are fundamentally about taking boolean values and conditionally executing code (usually to generate other values). In cases where all we need is a boolean value, *it is often simpler to write an expression to calculate the value directly, rather than using if statements*.

In our example, the if statement is redundant and can be simplified just by returning the value of the condition:

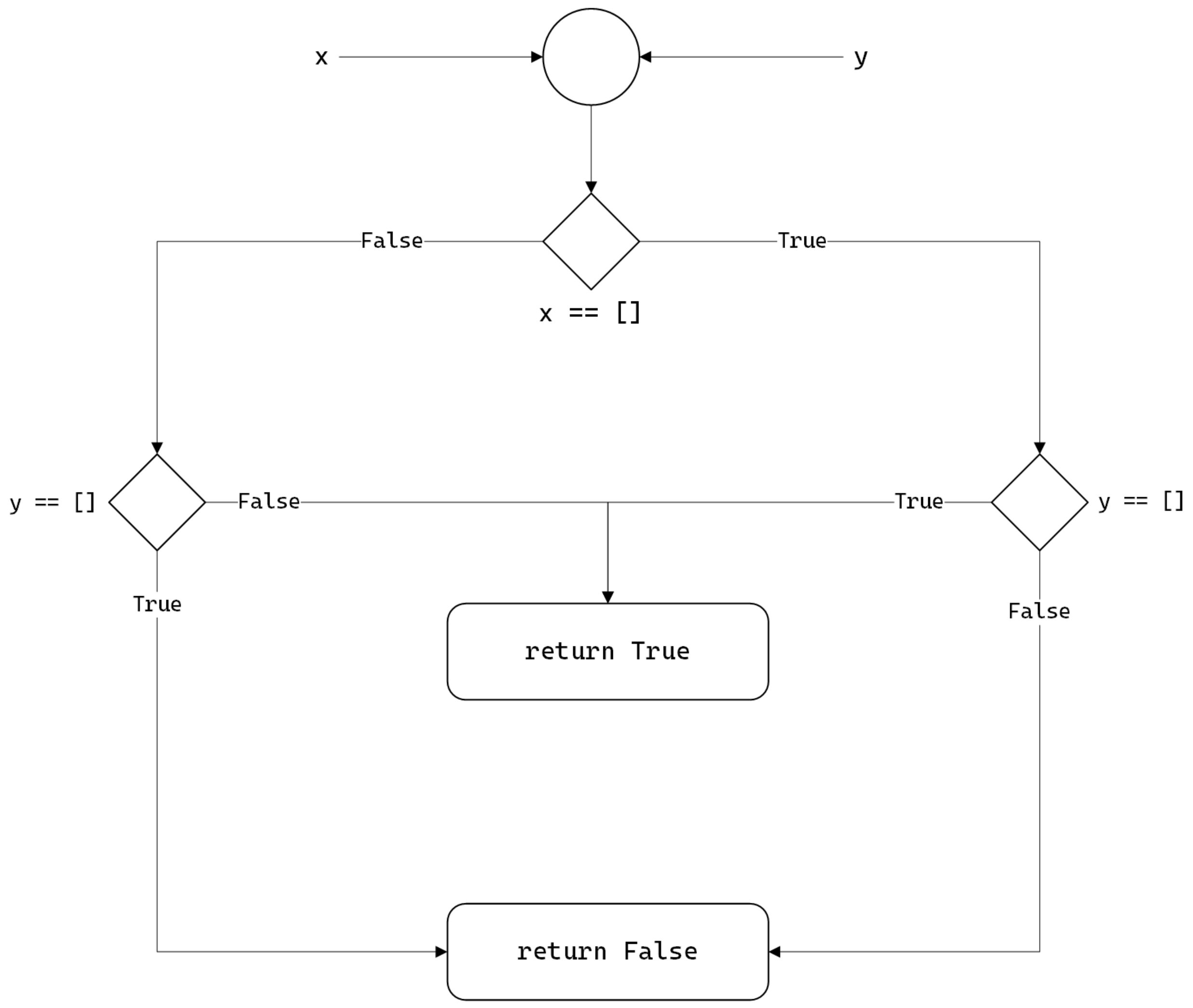
```python
def is_even(n: int) -> bool:
    """Return whether n is even (divisible by 2)."""
    return n % 2 == 0
```

Indeed, our earlier study of propositional logic should make us comfortable with the idea of treating booleans just like any other kind of value, and we should make full use of Python's logical operators `and`, `or`, and `not` to combine them.

Consider this more complex example with nested if statements:

```
 1  def mystery(x: lst, y: lst) -> bool:
 2      if x == []:
 3          if y == []:
 4              return True
 5          else:
 6              return False
 7      else:
 8          if y == []:
 9              return False
10          else:
11              return True
```

Here is a control flow diagram for this function, showing the four different possible execution paths.



To simplify this, we start with the first inner if statement on lines 3-6. This follows the same structure as our first example, and can be simplified to just `return y == []`.

The second inner if statement on lines 8-11 follows a similar structure, except that now the boolean that's returned is the negation of the if condition. So we can simplify this as `return not y == []`, which we can simplify further using the `!=` operator: `return y != []`.

So now we have this simplification of the function body:

```
def mystery(x: lst, y: lst) -> bool:
    if x == []:
        return y == []
    else:
        return y != []
```

But now how do we simplify this further? The idea here is to focus on the possible ways that `mystery` could return `True`. The if statement divides the inputs into two cases: when `x == []` and the if branch executes, and when `x != []` and the else branch executes. In the first case, when `x == []`, `mystery` returns the value of `y == []`. So one case for `mystery` returning `True` is when `x == [] and y == []`. Similarly, in the second case, when `x != []`, `mystery` returns `y != []`, and so the other case for `mystery` returning `True` is `x != [] and y != []`.

How should we combine these two cases? Because these are different cases, either one of them could occur, but we don't expect both of them to occur (since `x == []` and `x != []` can't both be true), and so we combine them using `or`:

```
def mystery(x: lst, y: lst) -> bool:
    return (x == [] and y == []) or (x != [] and y != [])
```

This simplification took a bit of work, but as a result we have a clearer picture of what this function does. We can illustrate this further by breaking up the nested expression using local variables with meaningful names.

```
def mystery(x: lst, y: lst) -> bool:
    both_empty = x == [] and y == []
    both_non_empty = x != [] and y != []
    return both_empty or both_non_empty
```

To check your understanding, try writing a docstring description for this function. You'll probably find it at least a little easier to do for this version than the original. And while this is still a relatively small example, the same principle will often apply in the future, and so be on the lookout for if statements that you can simplify in this way.[1]

[1] That said, this simplification won't always apply or be appropriate, depending on the complexity of the branches of the statement. We'll discuss this in more detail later.

Using if statements

`if` statements create branches in our code, allowing us to create more advanced functions. But more branches means more complexity because there are many possible paths that our function could take when called. To mitigate the complexity that comes with branching, we recommend two principles when working with if statements:

1. Prefer using a sequence of `elif`s rather than nested `if` statements. Overuse of nesting makes your code harder to understand, and can make the visual structure of your code more complex than necessary.
2. Write your conditions from most specific to most general. Order matters for these conditions, since they are checked one at a time in top-down order.